Place-and-Route, and Bitstream Generation

Christian Krieg

November 20, 2024

Welcome to this year's *Digital Integrated Circuits* course! Throughout this course, we will have to solve some assignments that will teach us fundamental concepts of integrated circuit design.

We will demonstrate digital integrated circuit design on a step-by-step simple example of a digital counter. As the name of this course suggests, our ultimate goal is to implement this counter as a digital integrated circuit. In the lab, we map this counter to a field-programmable gate array (FPGA) architecture, and run it on the board shown in Figure 1. The counter's functionality is executed on the FPGA. We will be able to control execution by pushing the buttons, and to visualize the counter value using the light-emitting diodes (LEDs). Because operating a hardware counter with buttons and LEDs may be inconvenient, we also will interface the hardware counter over a universal asynchronous receiver/transmitter (UART), so that we may be able to further process the counter value on a host computer. But this will all happen during the lab week at the end of the exercises.

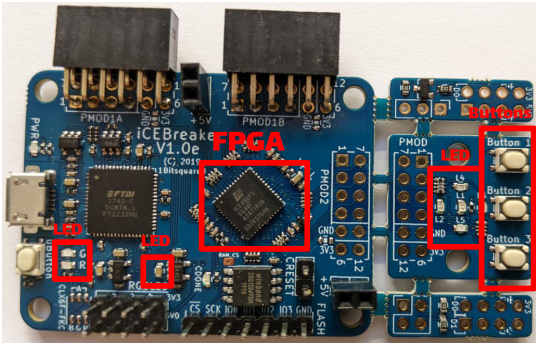

Figure 1: The *icebreaker* board, which will run our counter

Before we can bring our counter on hardware, it is important that we learn how to design digital circuits using state-of-the-art (and beyond) methods and tools, and to get a feeling for the results of our design decisions. We will draw attention on the results of the synthesis steps, which we will verify with several tools along the digital design flow.

Generally speaking, we perform the following steps:

1. Specify the counter's behavior
2. Model (implement) the counter in a hardware description language (HDL)
3. Verify the counter's implementation
4. Synthesize and optimize the counter's HDL model + verify
5. Map the counter to the target (FPGA) technology + verify
6. **Generate an FPGA bitstream + verify**
7. Configure the FPGA with the bitstream + verify
8. Run the counter on hardware

Our overall task is to build a synchronous counter that runs in two modes, based on a *mode* switch. The two modes are called *up* and *down*. In *up* mode, the counter increments the counter value by **5** at the rising edge of the clock signal, and in *down* mode, it decrements the counter value by **9** at the rising edge of the clock signal. The counter should implement synchronous reset behavior, such that it is set to an initial value of **-50** upon reset. The counter should never exceed a value of **235**, and should never go below a value of **-230** (the counter value sticks near the minimum/maximum value, until the counting direction changes). The counter value should never be **-11** (the counter should jump over this value by one increment/decrement).

Glossary

CSV Comma-separated values. 3

FPGA Field-programmable gate array. 1, 2, 4

HDL Hardware description language. 1

JSON JavaScript obejct notation. 2, 3

LED Light-emitting diode. 1

RTL Register transfer level. 2

STA Static timing analysis. 3, 4

UART Universal asynchronous receiver/transmitter. 1

Place and route your counter, encode the bitstream

We created several implementations for the iCE40 target technology, observing different resource utilizations for different synthesis options. Our main observation is that the designs that are synthesized from the gate-level representation consume less resources than those synthesized from the register transfer level (RTL) representation.

In this task, we will place and route these designs for an actual chip, creating a *bitstream*, and we will investigate the maximum speed at which we can operate them on actual hardware. We will learn to use 'nextpnr' to place and route our design. We will examine the basic options for 'nextpnr' to create a text-based bitstream and a placed-and-routed netlist. We will learn how a physical constraints file looks like, and we will have a look at the graphical interface of 'nextpnr'. We will investigate the maximum operating frequency for each counter implementation, and draw conclusions on the different implementations. We will obtain the following skills:

1. Use *nextpnr* to place and route our designs
2. Use *icetime* to perform static timing analysis
3. Interpreting the output of *icetime*
4. OPTIONAL: Use *bash* to automate everything

Placement and routing

In placement and routing, we map the technology-mapped netlist to the resources available on an actual chip (*placement*), and the logic is interconnected in a *routing* process. In this course, we use the *Icebreaker boarf* for evaluation, which incorporates a Lattice iCE40-UP5K FPGA. We use *nextpnr* to place and route our designs.

nextpnr accepts as input format a technology-mapped netlist in JavaScript obejct notation (JSON) format. *Yosys* supports JSON netlists, so the first step is to extend our synthesis script from Task 6 with a `write_json` command as demonstrated in Listing 1.

Listing 1: Yosys command to write a technology-mapped netlist in JSON format

```
[...]
design -load 1
synth_ice40
write_verilog -noexpr 1.v
write_json 1.json
design -save 1
[...]
```

We now call *nextpnr* on that netlist from the shell, as given in Listing 2.

Note the options in Table 1 that are passed to *nextpnr*.

Listing 2: Command to invoke *nextpnr* to place and route your counter for the iCE40-HX8K FPGA

```
nextpnr-ice40 --up5k --package sg48 --json 1.json --pcf counter.pcf --write 1.pnr --asc 1.asc
```

Table 1: Options passed to *nextpnr*

`--up5k`	Sets the device type
`--package sg48`	Sets the device package
`--json 1.json`	Specifies the input netlist
`--pcf counter.pcf`	Specifies the physical constraints file
`--write 1.pnr`	Specifies the file name for the placed-and-routed netlist
`--asc 1.asc`	Sets the file name of the output bitstream

The first two options set the device type and the package and are predetermined by the evaluation board. The third option specifies the input file we just created. The fourth option specifies the physical constraints for our design. In principle, we connect the top-level interface of our design to the pins of the chip. Listing 3 shows an example *.pcf* file that we use for this task. The fifth option specifies the filename to which the placed-and-routed netlist is written. Again, this will be a file in JSON format. The last argument specifies the name of the output file, which is a bitstream in ASCII format.

Listing 3: Sample constraints file for placement and routing (*counter.pcf*)

```
set_io clk 44
set_io rst 45
set_io mode 46

set_io cnt[0] 42
set_io cnt[1] 43
set_io cnt[2] 36
set_io cnt[3] 38
set_io cnt[4] 32
set_io cnt[5] 34
set_io cnt[6] 28
set_io cnt[7] 31
set_io cnt[8] 2
set_io cnt[9] 3
```

Voilá, we now have a placed-and-routed bitstream in ASCII format (given in file *1.asc*). We also created a placed-and-routed netlist that specifies the locations of the placed logic cells, and the routings by which these cells are interconnected. You can have a look at your placed and routed design by invoking *nextpnr* with the *–gui* option as given in Listing 4.

Note that you also could place and route your design using *nextpnr*'s graphical user interface. However, this requires several clicks, and will take you longer than just using the command line. It's up to you!

Timing analysis

We are now at a point where we can make statements about the timing behavior of our designs, because we now know the routing; together with the delays introduced by the logic cells we can calculate the critical path. The critical path is the path with the largest delay, which determines the maximum operation frequency. To calculate the critical path, we perform a static timing analysis (STA), for which we use *icetime* as given in Listing 5. Again, we provide information on the constraints, the device, and the package.

icetime will output something like Listing 6. Note the last line, where we get a timing estimate with the maximum operating frequency.

Our task is now to determine the timing for all four designs we created in Task 6. We can do it by hand, repeatedly performing the commands given in the above listings. But we can also use a shell script to do the work for us. Listing 7 shows two *bash* command lines to automate everything. In case you choose the automatic way: Make yourself familiar with the commands and understand what they are doing. Just in case you know somebody that speaks *bash*, and you don't: Ask them!

Interpreting timing results: Design space exploration

We are now at a point where we have available information on the utilized resources, and the timing of our counter implementations. It is now time to draw some conclusions on the data. Have a look at the utilized resources for all four

Listing 4: Command to invoke *nextpnr* to view your placed and routed design

```
nextpnr-ice40 --gui --json 1.pnr
```

Listing 5: Command to invoke *icetime* to perform an STA

```
icetime -p counter.pcf -d up5k -P sg48 1.asc
```

design variations we obtained in Task 6, and compare it to the timing information we obtained in this task. What do we observe? We definitely observe that our design and synthesis decisions have an impact on both the resources and the timing. What would you consider the best decisions? And why?

Let's extend our comma-separated values (CSV) file from the previous task (Table 2). What we see is that we have four implementations with different resource usage and maximum operating frequency. Remember: each implementation implements the expected behavior, but runs at different speeds, and requires different amounts of hardware resources. Isn't that amazing? So, it is now our task to identify the *best* implementation in order to turn it into real hardware.

But wait; what does "the best implementation" mean? Let's further simplify Table 2 by removing irrelevant data (that is equal for every implementation), and only consider the total cell counts instead of the individual counts, and we only consider the number of wire bits (and ignore the number of wires). What we get is a three-dimensional design space for our four counter implementations (Table 3). The three dimensions (or, parameters) of the design space are the number of bits, the number of cells, and the maximum operating frequency.

Each of the three parameters can be "best" for any of the counters, but the combinations of all parameters for one counter determines whether it is optimal. The concept of choosing one implementation over another depending on multiple parameters can be accomplished by identifying the set of *Pareto-efficient* implementations. We identify Pareto-efficient solutions as follows:

1. If a design is worse in all parameters than another design, then it is **not** Pareto-efficient.
2. If a design is equal in some (not all) and worse in the rest of the parameters, then it is **not** Pareto-efficient.
3. If a design has parameters that are better than any other design, then it is Pareto-efficient.
4. If a design has parameters that are better than in some Pareto-efficient designs, then it too is Pareto-efficient.

Having a look at Table 3, we note that there are two implementation meetings these requirements: implementations *2* and *4*. Figure 2 illustrates the design space described by Table 3 as a 3-D plot. The Pareto-efficient implementations are colored red.

It is now your task to identify the Pareto-optimal solutions among your four counter implementations!

Encode the bitstream

Congratulations! We are almost at the point, where we can configure our FPGA device, and run our counter on real hardware! There is one little step to do, which is to turn the ASCII-formatted bitstream into a binary bitstream, for which we use *icepack*. In the following, we will:

1. Use *icepack* to encode the bistream
2. Use *iceunpack* to decode the bitstream
3. Use *icebox_vlog* to extracted a Verilog netlist from the decoded bitstream
4. Use *SymbiYosys* to formally verify the counter's correct behavior

Table 2: Resources and timing results of our four counter implementations

Design	Wires	Bits	Public wires	Public bits	Total Cells	SB_LUT4	SB_DFFSR	SB_CARRY	Max. frequency
1	31	159	4	13	94	44	10	40	33.56 Mhz
2	33	137	4	13	54	44	10	0	34.5 Mhz
3	48	193	4	13	108	67	10	31	27.6 Mhz
4	42	138	4	13	65	55	10	0	41.61 Mhz

Listing 6: Sample output of *icetime*

```
// Reading input .pcf file..
// Reading input .asc file..
// Reading 8k chipdb file..
// Creating timing netlist..
// Timing estimate: 11.50 ns (86.98 MHz)
```

Listing 7: Sample *bash* script to automatically place and route all designs and perform STA

```
# Place and route all designs
for i in *.json; do nextpnr-ice40 --package sg48 --up5k --json $i --pcf counter.pcf --asc $i.asc; done

# Perform timing analysis and strip away all unnecessary output
for i in *.asc; do printf "`basename -s .json.asc $i`: "; icetime -p counter.pcf -d up5k -P sg48 $i |
    grep "Timing estimate" | sed -e "s/\/\/ Timing estimate: \(.*\) (\(.*\))/\2/g" ;done
```

Table 3: Design space spanned by our four counter implementations (visualized in Figure 2)

Design	Bits	Total Cells	Max. frequency
1	159	94	33.56 MHz
2	137	54	34.5 MHz
3	193	108	27.6 MHz
4	138	65	41.61 MHz

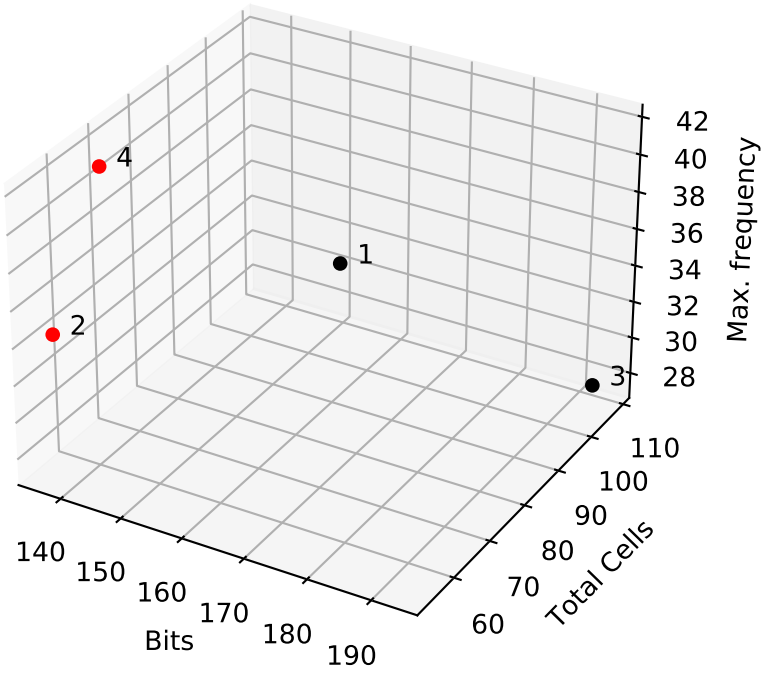

Figure 2: Design space visualized in a 3-D plot (Pareto-efficient solutions marked red)

Bitstream encoding

Take your Pareto-efficient design implementations, and encode them as given in Listing 8. Use the design numbers as filenames.

Listing 8: Example command to create a binary representation of the target bitstreams using *icepack*

```
icepack 4.json.asc 4.bin
```

Verification

We can still use the assertions we specified in Tasks 1 and 2 to verify our counter. All we have to do is to decode the bitstream again into the ASCII-based format using *iceunpack*, and turn the bitstream into a Verilog netlist again using *icebox_vlog*. We may then connect the counter to the formal testbench (within the file *tb_counter.sv*), and check the properties using *SymbiYosys*. (Listing 9) shows the command sequence to verify the pareto-optimal implementations identified in Table 3. Note that this may be different for you! Adjust the sequence according to your setting.

Listing 9: Example command sequence to extract a Verilog model from our bitstream, and to verify it using *SymbiYosys*

```
iceunpack 4.bin 4u.asc
icebox_vlog -p counter.pcf 4u.asc >4u.v
sby -f verify.sby
```

The SymbiYosys configuration used in Listing 9 is given in Listing 10. Adjust it according to your needs!

Listing 10: SymbiYosys configuration to verify the correct behavior of our bitstreams

```
[tasks]
counter
2
4

[options]
mode prove
expect pass

[engines]
smtbmc

[script]
read -define NO_ICE40_DEFAULT_ASSIGNMENTS
read -formal cells_sim.v
read -formal tb_counter.sv
counter: read -sv counter.v
2: read -formal 2u.v
4: read -formal 4u.v
verific -import tb_counter
prep -top tb_counter
counter: opt -full
counter: techmap -map map.v

[files]
tb_counter.sv
counter: counter.v
counter: map.v
2u.v
4u.v
--pycode-begin--
import subprocess
output(subprocess.run(["yosys-config", "--datdir/ice40/cells_sim.v"], capture_output=True).stdout.
    decode("utf-8"))
--pycode-end--
```

Submission instructions

Attach your *counter.v* and *map.v*.

Also, attach all Pareto-efficient bitstreams in binary format to your submission e-mail. The filenames for each bitstream must be the number of the design, with ".bin" as the file extension.